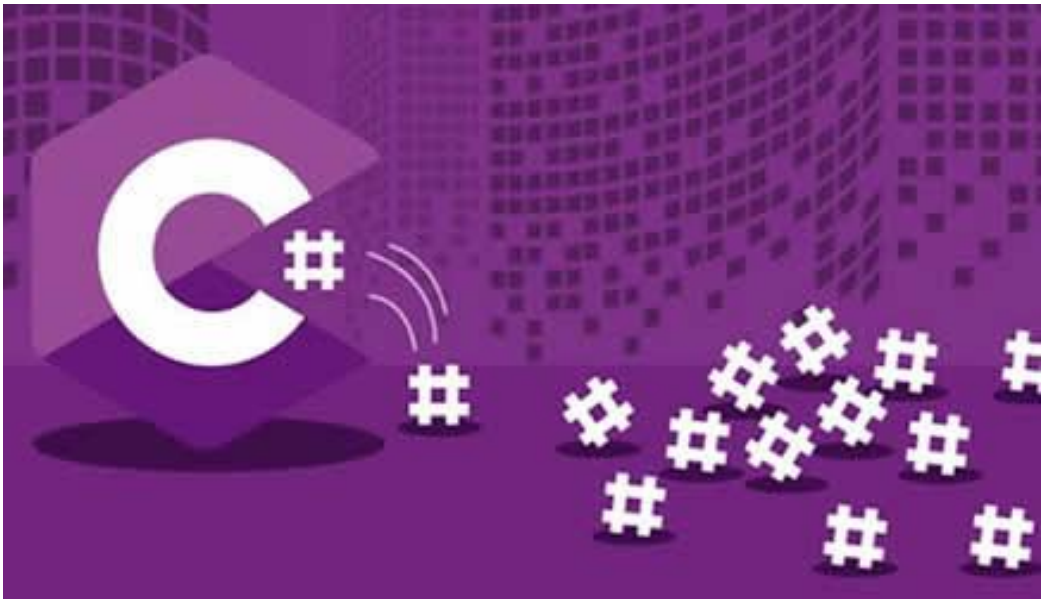


TABLES

- C# output
- C# comments
- C# Variables
- C# data types
- C# type casting
- C# user input
- C# Operators
- C# math
- C# strings
- C# Booleans
- C# If... Else
- C# switch
- C# While Boucle
- C# for loop
- C# Pause/Continue
- C# arrays

C# methods

- C# methods
- C# method parameters
- C# method overload

C# course

- C# POO
- C# Classes/Objects
- C# Class Members
- C# constructors
- C# Access Modifiers
- C# Properties
- C# legacy
- C# polymorphism
- C# Abstraction
- Interface C#
- C# enumerations
- C# files
- C# exception

Sortie C#

Pour générer des valeurs ou imprimer du texte en C#, vous pouvez utiliser la `WriteLine()`méthode :

```
Console.WriteLine("Hello World!");
```

La méthode d'écriture

Il existe également une `Write()`méthode similaire à `WriteLine()`.

La seule différence est qu'il n'insère pas de nouvelle ligne à la fin de la sortie :

Commentaires C#

Les commentaires peuvent être utilisés pour expliquer le code C# et le rendre plus lisible. Il peut également être utilisé pour empêcher l'exécution lors du test d'un code alternatif.

Commentaires sur une seule ligne

Les commentaires sur une seule ligne commencent par deux barres obliques (`//`).Tout texte entre `//`et la fin de la ligne est ignoré par C# (ne sera pas exécuté).Cet exemple utilise un commentaire sur une seule ligne avant une ligne de code :

```
// This is a comment

Console.WriteLine("Hello World!");
```

Commentaires multi-lignes C#

Les commentaires multilignes commencent par `/*`et se terminent par `*/`.

Tout texte entre `/*`et `*/`sera ignoré par C#.Cet exemple utilise un commentaire multiligne (un bloc de commentaire) pour expliquer le code :

```
/* The code below will print the words Hello World

to the screen, and it is amazing */

Console.WriteLine("Hello World!");
```

Variables C#

Les variables sont des conteneurs pour stocker des valeurs de données.

En C#, il existe différents **types** de variables (définies avec différents mots clés), par exemple :

- `int`- stocke des nombres entiers (nombres entiers), sans décimales, comme 123 ou -123
- `double`- stocke les nombres à virgule flottante, avec des décimales, telles que 19,99 ou -19,99
- `char`- stocke des caractères uniques, tels que 'a' ou 'B'. Les valeurs char sont entourées de guillemets simples
- `string`- stocke du texte, tel que "Hello World". Les valeurs de chaîne sont entourées de guillemets doubles
- `bool`- stocke les valeurs avec deux états : vrai ou faux

Créez une variable appelée **nom** de type `string`et affectez-lui la valeur " **John** " :

```
string name = "John";

Console.WriteLine(name);
```

Créez une variable appelée **myNum** de type `int`et affectez-lui la valeur **15** :

```
int myNum = 15;
```

```
Console.WriteLine(myNum);
```

Vous pouvez également déclarer une variable sans affecter la valeur, et affecter la valeur plus tard :

Exemple

```
int myNum;

myNum = 15;

Console.WriteLine(myNum);
```

Notez que si vous affectez une nouvelle valeur à une variable existante, elle écrasera la valeur précédente :

Exemple

Remplacez la valeur de `myNum`20 :

```
int myNum = 15;

myNum = 20; // myNum is now 20

Console.WriteLine(myNum);
```

Une démonstration de la façon de déclarer des variables d'autres types :

Exemple

```
int myNum = 5;

double myDoubleNum = 5.99D;

char myLetter = 'D';

bool myBool = true;

string myText = "Hello";
```

Constantes

Si vous ne voulez pas que d'autres (ou vous-même) remplacent les valeurs existantes, vous pouvez ajouter le mot- `const`clé devant le type de variable.

Cela déclarera la variable comme "constante", ce qui signifie non modifiable et en lecture seule :

```
const int myNum = 15;

myNum = 20; // error
```

Le `const`mot-clé est utile lorsque vous souhaitez qu'une variable stocke toujours la même valeur, afin que les autres (ou vous-même) ne gâchent pas votre code. Un exemple qui est souvent appelé une constante est PI (3.14159...).

Remarque : Vous ne pouvez pas déclarer une variable constante sans affecter la valeur. Si vous le faites, une erreur se convient : un champ const nécessite qu'une valeur soit fournie .

Variables d'affichage

La `WriteLine()`méthode est souvent utilisée pour afficher les valeurs des variables dans la fenêtre de la console.

Pour combiner à la fois du texte et une variable, utilisez le `+`caractère :

```
string name = "John";
```

```
Console.WriteLine("Hello " + name);
```

Toutes les variables C# doivent être **identifiées** par **des noms uniques** .

Ces noms uniques sont appelés **identificateurs** .

Les identifiants peuvent être des noms courts (comme x et y) ou des noms plus descriptifs (age, sum, totalVolume).

Remarque : Il est recommandé d'utiliser des noms descriptifs afin de créer un code compréhensible et maintenable :

```
// Good
int minutesPerHour = 60;


// OK, but not so easy to understand what m actually is
int m = 60;
```

Les règles générales pour nommer les variables sont :

- Les noms peuvent contenir des lettres, des chiffres et le caractère de soulignement (_)
- Les noms doivent commencer par une lettre
- Les noms doivent commencer par une lettre minuscule et ne peuvent pas contenir d'espaces
- Les noms sont sensibles à la casse ("myVar" et "myvar" sont des variables différentes)
- Les mots réservés (comme les mots-clés C#, tels que `int`ou `double`) ne peuvent pas être utilisés comme noms

Types de données C#

Comme expliqué dans le chapitre sur les variables, une variable en C# doit être un type de données spécifié :

```
int myNum = 5;               // Integer (whole number)
double myDoubleNum = 5.99D;  // Floating point number
char myLetter = 'D';         // Character
bool myBool = true;          // Boolean
string myText = "Hello";     // String
```

Un type de données spécifie la taille et le type des valeurs de variable.

Il est important d'utiliser le type de données correct pour la variable correspondante ; pour éviter les erreurs, pour gagner du temps et de la mémoire, mais cela rendra également votre code plus maintenable et lisible. Les types de données les plus courants sont :

Data Type	Size	Description
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
bool	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter, surrounded by single quotes
string	2 bytes per character	Stores a sequence of characters, surrounded by double quotes

Nombres

Les types de nombres sont divisés en deux groupes :

Les types entiers stockent des nombres entiers, positifs ou négatifs (tels que 123 ou -456), sans décimales. Les types valides sont `int` et `long`. Le type à utiliser dépend de la valeur numérique.

Les types à virgule flottante représentent des nombres avec une partie fractionnaire, contenant une ou plusieurs décimales. Les types valides sont `float`et `double`.

Même s'il existe de nombreux types numériques en C#, les plus utilisés pour les nombres sont `int`(pour les nombres entiers) et `double`(pour les nombres à virgule flottante). Cependant, nous les décrirons tous au fur et à mesure de votre lecture.

Types entiers

Int

Le `int`type de données peut stocker des nombres entiers de -2147483648 à 2147483647. En général, et dans notre tutoriel, le `int`type de données est le type de données préféré lorsque nous créons des variables avec une valeur numérique.

```
int myNum = 100000;

Console.WriteLine(myNum);
```

Long

Le `long`type de données peut stocker des nombres entiers de -9223372036854775808 à 9223372036854775807. Ceci est utilisé lorsque `int`n'est pas assez grand pour stocker la valeur. Notez que vous devez terminer la valeur par un "L":

```
long myNum = 15000000000L;

Console.WriteLine(myNum);
```

Types à virgule flottante

Vous devez utiliser un type à virgule flottante chaque fois que vous avez besoin d'un nombre avec une décimale, comme 9,99 ou 3,14515.

Les types de données `float`et `double`peuvent stocker des nombres fractionnaires. Notez que vous devez terminer la valeur par un "F" pour les flottants et un "D" pour les doubles :

```
float myNum = 5.75F;

Console.WriteLine(myNum);
```

Double exemple

```
double myNum = 19.99D;

Console.WriteLine(myNum);
```

Utilisez `float`ou `double`?

La **précision** d'une valeur à virgule flottante indique le nombre de chiffres que la valeur peut avoir après la virgule décimale. La précision de `float`n'est que de six ou sept chiffres décimaux, tandis que `double`les variables ont une précision d'environ 15 chiffres. Par conséquent, il est plus sûr de l'utiliser `double`pour la plupart des calculs.

Numéros scientifiques

Un nombre à virgule flottante peut aussi être un nombre scientifique avec un "e" pour indiquer la puissance de 10 :

```
float f1 = 35e3F;

double d1 = 12E4D;

Console.WriteLine(f1);

Console.WriteLine(d1);
```

Coulée de type C#

Le cast de type consiste à attribuer une valeur d'un type de données à un autre type.

En C#, il existe deux types de casting :

- **Implicit Casting** (automatiquement) - conversion d'un type plus petit en une taille de type plus grande
  `char-> int-> long-> float->double`
- **Casting explicite** (manuellement) - conversion d'un type plus grand en un type de taille plus petite
  `double-> float-> long-> int->char`

Casting implicite

Le transtypage implicite est effectué automatiquement lors du passage d'un type de taille inférieure à un type de taille supérieure :

```
int myInt = 9;

double myDouble = myInt;       // Automatic casting: int to double



Console.WriteLine(myInt);      // Outputs 9

Console.WriteLine(myDouble);   // Outputs 9
```

Casting explicite

Le casting explicite doit être fait manuellement en plaçant le type entre parenthèses devant la valeur :

```
double myDouble = 9.78;
```

```
int myInt = (int) myDouble;    // Manual casting: double to int


Console.WriteLine(myDouble);   // Outputs 9.78

Console.WriteLine(myInt);      // Outputs 9
```

Méthodes de conversion de types

Il est également possible de convertir explicitement les types de données à l'aide de méthodes intégrées, telles que `Convert.ToBoolean`, `Convert.ToDouble`, `Convert.ToString`, `Convert.ToInt32( int)` et `Convert.ToInt64( long)` :

```
int myInt = 10;

double myDouble = 5.25;

bool myBool = true;


Console.WriteLine(Convert.ToString(myInt));    // convert int to string

Console.WriteLine(Convert.ToDouble(myInt));    // convert int to double

Console.WriteLine(Convert.ToInt32(myDouble));  // convert double to int

Console.WriteLine(Convert.ToString(myBool));   // convert bool to string
```

Pourquoi Convertir ?

Plusieurs fois, il n'y a pas besoin de conversion de type. Mais parfois il faut le faire. Jetez un œil au chapitre suivant, lorsque vous travaillez avec [l'entrée utilisateur] , pour en voir un exemple.

Entrée utilisateur C#

Obtenir l'entrée de l'utilisateur

Vous avez déjà appris que `Console.WriteLine()` est utilisé pour sortir (imprimer) des valeurs. Maintenant, nous allons utiliser `Console.ReadLine()` pour obtenir l'entrée de l'utilisateur.

Dans l'exemple suivant, l'utilisateur peut entrer son nom d'utilisateur, qui est stocké dans la variable `userName`. Puis on imprime la valeur de `userName`:

```
// Type your username and press enter

Console.WriteLine("Enter username:");



// Create a string variable and get user input from the keyboard and store
it in the variable

string userName = Console.ReadLine();



// Print the value of the variable (userName), which will display the
input value

Console.WriteLine("Username is: " + userName);
```

Saisie utilisateur et nombres

La `Console.ReadLine()` méthode retourne un `string`. Par conséquent, vous ne pouvez pas obtenir d'informations à partir d'un autre type de données, tel que `int`. Le programme suivant provoquera une erreur :

```
Console.WriteLine("Enter your age:");

int age = Console.ReadLine();

Console.WriteLine("Your age is: " + age);
```

Le message d'erreur ressemblera à ceci :

```
Cannot implicitly convert type 'string' to 'int'
```

Comme le message d'erreur l'indique, vous ne pouvez pas convertir implicitement le type 'string' en 'int'.

Heureusement, pour vous, vous venez d'apprendre du chapitre précédent (Type Casting) , que vous pouvez convertir n'importe quel type explicitement, en utilisant l'une des `Convert.To`méthodes :

```
Console.WriteLine("Enter your age:");

int age = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Your age is: " + age);
```

Remarque : Si vous entrez une mauvaise entrée (par exemple, du texte dans une entrée numérique), vous obtiendrez un message d'exception/d'erreur (comme System.FormatException : 'La chaîne d'entrée n'était pas dans un format correct.').

Vous en apprendrez plus sur les exceptions et sur la façon de gérer les erreurs dans un chapitre ultérieur.

Opérateurs C#

Les opérateurs

Les opérateurs sont utilisés pour effectuer des opérations sur des variables et des valeurs.

Dans l'exemple ci-dessous, nous utilisons l' `+` **opérateur** pour additionner deux valeurs :

```
int x = 100 + 50;
```

Bien que l' `+`opérateur soit souvent utilisé pour additionner deux valeurs, comme dans l'exemple ci-dessus, il peut également être utilisé pour additionner une variable et une valeur, ou une variable et une autre variable :

```
int sum1 = 100 + 50;         // 150 (100 + 50)

int sum2 = sum1 + 250;       // 400 (150 + 250)

int sum3 = sum2 + sum2;      // 800 (400 + 400)
```

Opérateurs arithmétiques

Les opérateurs arithmétiques sont utilisés pour effectuer des opérations mathématiques courantes :

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	x++
--	Decrement	Decreases the value of a variable by 1	x--

Opérateurs d'affectation

Les opérateurs d'affectation sont utilisés pour affecter des valeurs aux variables.

Dans l'exemple ci-dessous, nous utilisons l' opérateur d' **affectation**`=` ( ) pour affecter la valeur **10** à une variable appelée **x** :

```
int x = 10;
```

L' opérateur **d'affectation d'addition**`+=` ( ) ajoute une valeur à une variable :

```
int x = 10;

x += 5;
```

Une liste de tous les opérateurs d'affectation :

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Opérateurs de comparaison

Les opérateurs de comparaison sont utilisés pour comparer deux valeurs :

Remarque : La valeur de retour d'une comparaison est soit `True`ou `False`.

Dans l'exemple suivant, nous utilisons l' **opérateur supérieur à** (`>`) pour savoir si 5 est supérieur à 3 :

```
int x = 5;

int y = 3;

Console.WriteLine(x > y); // returns True because 5 is greater than 3
```

Une liste de tous les opérateurs de comparaison :

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Opérateurs logiques

Les opérateurs logiques sont utilisés pour déterminer la logique entre les variables ou les valeurs :

Operator	Name	Description	Example
&&	Logical and	Returns True if both statements are true	x < 5 && x < 10
||	Logical or	Returns True if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns False if the result is true	!(x < 5 && x < 10)

Mathématiques C#

La classe C # Math a de nombreuses méthodes qui vous permettent d'effectuer des tâches mathématiques sur des nombres.

Math.Max(*x,y*)

La méthode peut être utilisée pour trouver la valeur la plus élevée de *x* et *y* :`Math.Max(x,y)`

```
Math.Max(5, 10);
```

Math.Min(*x,y*)

La méthode peut être utilisée pour trouver la valeur la plus basse de *x* et *y* :`Math.Min(x,y)`

```
Math.Min(5, 10);
```

Math.Sqrt(*x*)

La méthode renvoie la racine carrée de *x* :`Math.Sqrt(x)`

```
Math.Sqrt(64);
```

Chaînes C#

Les chaînes sont utilisées pour stocker du texte.

Une `string`variable contient une collection de caractères entourés de guillemets :

Créez une variable de type `string`et affectez-lui une valeur :

```
string greeting = "Hello";
```

Une variable chaîne peut contenir plusieurs mots, si vous voulez :

```
string greeting2 = "Nice to meet you!";
```

Longueur de chaine

Une chaîne en C # est en fait un objet, qui contient des propriétés et des méthodes pouvant effectuer certaines opérations sur des chaînes. Par exemple, la longueur d'une chaîne peut être trouvée avec la `Length`propriété :

```
string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

Console.WriteLine("The length of the txt string is: " + txt.Length);
```

Autres méthodes

Il existe de nombreuses méthodes de chaîne disponibles, par exemple `ToUpper()`and `ToLower()`, qui renvoie une copie de la chaîne convertie en majuscule ou en minuscule :

```
string txt = "Hello World";

Console.WriteLine(txt.ToUpper());   // Outputs "HELLO WORLD"

Console.WriteLine(txt.ToLower());   // Outputs "hello world"
```

Concaténation de chaînes

L' `+`opérateur peut être utilisé entre les chaînes pour les combiner. C'est ce qu'on appelle **la concaténation** :

```
string firstName = "John ";

string lastName = "Doe";

string name = firstName + lastName;

Console.WriteLine(name);
```

Notez que nous avons ajouté un espace après "John" pour créer un espace entre firstName et lastName à l'impression.

Vous pouvez également utiliser la `string.Concat()`méthode pour concaténer deux chaînes :

```
string firstName = "John ";

string lastName = "Doe";

string name = string.Concat(firstName, lastName);

Console.WriteLine(name);
```

Ajouter des nombres et des chaînes

ATTENTION!

C# utilise l'opérateur + pour l'addition et la concaténation.

N'oubliez pas : les nombres s'additionnent. Les chaînes sont concaténées.

Si vous additionnez deux nombres, le résultat sera un nombre :

```
int x = 10;

int y = 20;

int z = x + y;  // z will be 30 (an integer/number)
```

Si vous ajoutez deux chaînes, le résultat sera une concaténation de chaînes :

```
string x = "10";

string y = "20";

string z = x + y;  // z will be 1020 (a string)
```

Interpolation de chaîne

Une autre option de concaténation de chaînes est **l'interpolation** de chaînes , qui remplace les valeurs des variables par des espaces réservés dans une chaîne. Notez que vous n'avez pas à vous soucier des espaces, comme avec la concaténation :

```
string firstName = "John";

string lastName = "Doe";

string name = $"My full name is: {firstName} {lastName}";
```

```
Console.WriteLine(name);
```

Notez également que vous devez utiliser le signe dollar (`$`) lorsque vous utilisez la méthode d'interpolation de chaîne.

L'interpolation de chaîne a été introduite dans C# version 6.

Chaînes d'accès

Vous pouvez accéder aux caractères d'une chaîne en vous référant à son numéro d'index entre crochets `[]`.

Cet exemple imprime le **premier caractère** de **myString** :

```
string myString = "Hello";

Console.WriteLine(myString[0]);  // Outputs "H"
```

Remarque : Les index de chaîne commencent par 0 : [0] est le premier caractère. [1] est le deuxième caractère, etc.

```
string myString = "Hello";

Console.WriteLine(myString[1]);  // Outputs "e"
```

Vous pouvez également trouver la position d'index d'un caractère spécifique dans une chaîne, en utilisant la `IndexOf()`méthode :

```
string myString = "Hello";

Console.WriteLine(myString.IndexOf("e"));  // Outputs "1"
```

Une autre méthode utile est `Substring()`, qui extrait les caractères d'une chaîne, à partir de la position/index de caractère spécifié, et renvoie une nouvelle chaîne. Cette méthode est souvent utilisée avec `IndexOf()`pour obtenir la position de caractère spécifique :

```
// Full name

string name = "John Doe";


// Location of the letter D

int charPos = name.IndexOf("D");


// Get last name

string lastName = name.Substring(charPos);


// Print the result

Console.WriteLine(lastName);
```

Chaînes - Caractères spéciaux

Étant donné que les chaînes doivent être écrites entre guillemets, C# comprendra mal cette chaîne et générera une erreur :

```
string txt = "We are the so-called "Vikings" from the north.";
```

La solution pour éviter ce problème est d'utiliser le **caractère d'échappement antislash** .

Le caractère d'échappement barre oblique inverse (`\`) transforme les caractères spéciaux en caractères de chaîne :

Escape character	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

La séquence `\"` insère un guillemet double dans une chaîne :

```
string txt = "We are the so-called \"Vikings\" from the north.";
```

La séquence `\'` insère un guillemet simple dans une chaîne :

```
string txt = "It\'s alright.";
```

La séquence `\\` insère une seule barre oblique inverse dans une chaîne :

```
string txt = "The character \\ is called backslash.";
```

D'autres caractères d'échappement utiles en C# sont :

Code	Result
\n	New Line
\t	Tab
\b	Backspace

Booléens C#

Très souvent, en programmation, vous aurez besoin d'un type de données qui ne peut avoir qu'une des deux valeurs, comme :

- OUI NON
- ALLUMÉ ÉTEINT
- VRAI FAUX

Pour cela, C# a un `bool`type de données, qui peut prendre les valeurs `true`ou `false`.

Valeurs booléennes

Un type booléen est déclaré avec le `bool`mot clé et ne peut prendre que les valeurs `true`ou `false`:

```
bool isCSharpFun = true;

bool isFishTasty = false;

Console.WriteLine(isCSharpFun);   // Outputs True

Console.WriteLine(isFishTasty);   // Outputs False
```

Cependant, il est plus courant de renvoyer des valeurs booléennes à partir d'expressions booléennes, pour les tests conditionnels (voir ci-dessous).

Expression booléenne

Une **expression booléenne** est une expression C# qui renvoie une valeur booléenne : `True`ou `False`.

Vous pouvez utiliser un opérateur de comparaison, tel que l'opérateur **supérieur à** (`>`) pour savoir si une expression (ou une variable) est vraie :

```
int x = 10;

int y = 9;

Console.WriteLine(x > y); // returns True, because 10 is higher than 9
```

Ou encore plus simple :

```
Console.WriteLine(10 > 9); // returns True, because 10 is higher than 9
```

Dans les exemples ci-dessous, nous utilisons l' opérateur **égal à** (`==`) pour évaluer une expression :

```
int x = 10;

Console.WriteLine(x == 10); // returns True, because the value of x is
equal to 10
```

```
Console.WriteLine(10 == 15); // returns False, because 10 is not equal to
15
```

La valeur booléenne d'une expression est la base de toutes les comparaisons et conditions C#.

C# Si ... Sinon

Conditions C# et instructions If

C# prend en charge les conditions logiques habituelles des mathématiques :

- Inférieur à : `a < b`
- Inférieur ou égal à : `a <= b`
- Supérieur à : `a > b`
- Supérieur ou égal à : `a >= b`
- Égal à `a == b`

- Différent de : `a != b`

Vous pouvez utiliser ces conditions pour effectuer différentes actions pour différentes décisions.

C# contient les instructions conditionnelles suivantes :

- Utiliser `if`pour spécifier un bloc de code à exécuter, si une condition spécifiée est vraie
- Sert `else`à spécifier un bloc de code à exécuter, si la même condition est fausse
- Permet `else if`de spécifier une nouvelle condition à tester, si la première condition est fausse
- Utilisez `switch`pour spécifier de nombreux blocs de code alternatifs à exécuter

L'instruction if

Utilisez l' `if`instruction pour spécifier un bloc de code C# à exécuter si une condition est `True`.

```
if (condition)
{
  // block of code to be executed if the condition is True
}
```

Notez que `if`c'est en lettres minuscules. Les lettres majuscules (If ou IF) généreront une erreur.Dans l'exemple ci-dessous, nous testons deux valeurs pour savoir si 20 est supérieur à 18. Si la condition est `True`, imprimez du texte :

```
if (20 > 18)
{
  Console.WriteLine("20 is greater than 18");
}
```

On peut aussi tester des variables :

```
int x = 20;

int y = 18;

if (x > y)

{

  Console.WriteLine("x is greater than y");

}
```

Exemple expliqué

Dans l'exemple ci-dessus, nous utilisons deux variables, **x** et **y** , pour tester si x est supérieur à y (à l'aide de l' `>`opérateur). Comme x est 20, et y est 18, et nous savons que 20 est supérieur à 18, nous affichons à l'écran que "x est supérieur à y".

La déclaration d'autre

Utilisez l' `else`instruction pour spécifier un bloc de code à exécuter si la condition est `False`.

```
if (condition)

{

  // block of code to be executed if the condition is True

}

else

{

  // block of code to be executed if the condition is False

}
```

```
int time = 20;
```

```
if (time < 18)
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
// Outputs "Good evening."
```

Exemple expliqué

Dans l'exemple ci-dessus, le temps (20) est supérieur à 18, donc la condition est `False`. Pour cette raison, nous passons à la `else`condition et imprimons à l'écran "Bonsoir". Si l'heure était inférieure à 18, le programme imprimerait "Bonne journée".

L'instruction else if

Utilisez l' `else if`instruction pour spécifier une nouvelle condition si la première condition est `False`.

```
if (condition1)
{
  // block of code to be executed if condition1 is True
}
```

```
else if (condition2)

{

  // block of code to be executed if the condition1 is false and
condition2 is True

}

else

{

  // block of code to be executed if the condition1 is false and
condition2 is False

}
```

```
int time = 22;

if (time < 10)

{

  Console.WriteLine("Good morning.");

}

else if (time < 20)

{

  Console.WriteLine("Good day.");

}

else

{

  Console.WriteLine("Good evening.");

}

// Outputs "Good evening."
```

Exemple expliqué

Dans l'exemple ci-dessus, le temps (22) est supérieur à 10, donc la **première condition** est `False`. La condition suivante, dans la `else if`déclaration, est également `False`, nous passons donc à la `else` condition
puisque **condition1** et **condition2** sont les deux `False`- et affichons à l'écran "Bonsoir".

Cependant, si l'heure était 14h, notre programme afficherait "Bonne journée".

Abréviation If...Else (opérateur ternaire)

Il existe également un raccourci if else, connu sous le nom d' **opérateur ternaire** car il se compose de trois opérandes. Il peut être utilisé pour remplacer plusieurs lignes de code par une seule ligne. Il est souvent utilisé pour remplacer les instructions simples if else :

```
variable = (condition) ? expressionTrue :  expressionFalse;
```

Au lieu d'écrire :

```
int time = 20;
if (time < 18)
{
  Console.WriteLine("Good day.");
}
else
{
  Console.WriteLine("Good evening.");
}
```

Vous pouvez simplement écrire :

```
int time = 20;

string result = (time < 18) ? "Good day." : "Good evening.";

Console.WriteLine(result);
```

Instructions de commutation C#

Utilisez l' `switch`instruction pour sélectionner l'un des nombreux blocs de code à exécuter.

```
switch(expression)

{

  case x:

    // code block

    break;

  case y:

    // code block

    break;

  default:

    // code block

    break;

}
```

Voilà comment cela fonctionne:

- L' `switch`expression est évaluée une fois
- La valeur de l'expression est comparée aux valeurs de chaque`case`
- S'il y a correspondance, le bloc de code associé est exécuté
- Les mots clés `break`et `default`seront décrits plus loin dans ce chapitre

L'exemple ci-dessous utilise le numéro du jour de la semaine pour calculer le nom du jour de la semaine :

```
int day = 4;

switch (day)

{

  case 1:

    Console.WriteLine("Monday");

    break;

  case 2:

    Console.WriteLine("Tuesday");

    break;

  case 3:

    Console.WriteLine("Wednesday");

    break;

  case 4:

    Console.WriteLine("Thursday");

    break;

  case 5:

    Console.WriteLine("Friday");

    break;

  case 6:

    Console.WriteLine("Saturday");
```

```
    break;
  case 7:
    Console.WriteLine("Sunday");
    break;
}
// Outputs "Thursday" (day 4)
```

La pause

Lorsque C # atteint un mot- `break` clé, il sort du bloc de commutation.

Cela arrêtera l'exécution de plus de code et de tests de cas à l'intérieur du bloc.

Lorsqu'une correspondance est trouvée et que le travail est terminé, il est temps de faire une pause. Il n'y a pas besoin de plus de tests.

Un break peut faire gagner beaucoup de temps d'exécution car il "ignore" l'exécution de tout le reste du code dans le bloc switch.

Le mot clé par défaut

Le `default`mot-clé est facultatif et spécifie du code à exécuter s'il n'y a pas de correspondance de casse :

```
int day = 4;
switch (day)
{
  case 6:
    Console.WriteLine("Today is Saturday.");
    break;
```

```
  case 7:
    Console.WriteLine("Today is Sunday.");
    break;
  default:
    Console.WriteLine("Looking forward to the Weekend.");
    break;
}
// Outputs "Looking forward to the Weekend."
```

Boucles

Les boucles peuvent exécuter un bloc de code tant qu'une condition spécifiée est atteinte.

Les boucles sont pratiques car elles permettent de gagner du temps, de réduire les erreurs et de rendre le code plus lisible.

C# While Boucle

La `while`boucle parcourt un bloc de code tant qu'une condition spécifiée est `True` :

```
while (condition)
{
  // code block to be executed
}
```

Remarque : N'oubliez pas d'augmenter la variable utilisée dans la condition, sinon la boucle ne se terminera jamais !

La boucle faire/pendant que

La `do/while`boucle est une variante de la `while`boucle. Cette boucle exécutera le bloc de code une fois, avant de vérifier si la condition est vraie, puis elle répétera la boucle tant que la condition est vraie.

```
do

{

  // code block to be executed

}

while (condition);
```

L'exemple ci-dessous utilise une `do/while`boucle. La boucle sera toujours exécutée au moins une fois, même si la condition est fausse, car le bloc de code est exécuté avant que la condition ne soit testée :

```
int i = 0;
do

{

  Console.WriteLine(i);

  i++;

}

while (i < 5);
```

N'oubliez pas d'augmenter la variable utilisée dans la condition, sinon la boucle ne se terminera jamais !

C# pour la boucle

Lorsque vous savez exactement combien de fois vous voulez parcourir un bloc de code, utilisez la `for`boucle au lieu d'une `while`boucle :

```
for (statement 1; statement 2; statement 3)
{
  // code block to be executed
}
```

L' instruction 1 est exécutée (une fois) avant l'exécution du bloc de code.

L' instruction 2 définit la condition d'exécution du bloc de code.

L' instruction 3 est exécutée (à chaque fois) après l'exécution du bloc de code.

L'exemple ci-dessous imprimera les chiffres de 0 à 4 :

```
for (int i = 0; i < 5; i++)
{
  Console.WriteLine(i);
}
```

Exemple expliqué

L'instruction 1 définit une variable avant le début de la boucle (`int i = 0`).

L'instruction 2 définit la condition d'exécution de la boucle (`i`doit être inférieur à `5`). Si la condition est `true`, la boucle recommencera, si c'est `false`, la boucle se terminera.

L' instruction 3 augmente une valeur (`i++`) chaque fois que le bloc de code dans la boucle a été exécuté.

Un autre exemple

Cet exemple n'imprimera que les valeurs paires entre 0 et 10 :

```
for (int i = 0; i <= 10; i = i + 2)
{
  Console.WriteLine(i);
}
```

La boucle foreach

Il existe également une `foreach`boucle, qui est utilisée exclusivement pour parcourir les éléments d'un **tableau** :

```
foreach (type variableName in arrayName)
{
  // code block to be executed
}
```

L'exemple suivant affiche tous les éléments du tableau **cars**`foreach` à l'aide d'une boucle :

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
foreach (string i in cars)
{
  Console.WriteLine(i);
}
```

Remarque : Ne vous inquiétez pas si vous ne comprenez pas l'exemple ci-dessus. Vous en apprendrez plus sur les tableaux dans le chapitre Tableaux C# .

Coupure C#

Vous avez déjà vu l' `break`instruction utilisée dans un chapitre précédent de ce didacticiel. Il a été utilisé pour "sauter" d'une `switch`déclaration.

L' `break`instruction peut également être utilisée pour sortir d'une **boucle** .

Cet exemple sort de la boucle lorsque `i`est égal à `4`:

```
for (int i = 0; i < 10; i++)
{
  if (i == 4)
  {
    break;
  }
  Console.WriteLine(i);
}
```

C# Continuer

L' `continue`instruction interrompt une itération (dans la boucle), si une condition spécifiée se produit, et continue avec l'itération suivante dans la boucle.

Cet exemple ignore la valeur de`4` :

```
for (int i = 0; i < 10; i++)
```

```
{
  if (i == 4)
  {
    continue;
  }
  Console.WriteLine(i);
}
```

C# Arrays

Créer un tableau

Les tableaux sont utilisés pour stocker plusieurs valeurs dans une seule variable, au lieu de déclarer des variables distinctes pour chaque valeur.

Pour déclarer un tableau, définissez le type de la variable **entre crochets** :

```
string[] cars;
```

Nous avons maintenant déclaré une variable qui contient un tableau de chaînes.

Pour y insérer des valeurs, nous pouvons utiliser un tableau littéral - placez les valeurs dans une liste séparée par des virgules, à l'intérieur d'accolades :

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

Pour créer un tableau d'entiers, vous pouvez écrire :

```
int[] myNum = {10, 20, 30, 40};
```

Accéder aux éléments d'un tableau

Vous accédez à un élément de tableau en vous référant au numéro d'index.

Cette instruction accède à la valeur du premier élément dans **cars** :

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

Console.WriteLine(cars[0]);

// Outputs Volvo
```

Remarque : Les index de tableau commencent par 0 : [0] est le premier élément. [1] est le deuxième élément, etc.

Modifier un élément de tableau

Pour modifier la valeur d'un élément spécifique, reportez-vous au numéro d'index :

```
cars[0] = "Opel";
```

Exemple

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

cars[0] = "Opel";

Console.WriteLine(cars[0]);

// Now outputs Opel instead of Volvo
```

Longueur du tableau

Pour connaître le nombre d'éléments d'un tableau, utilisez la `Length`propriété :

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

Console.WriteLine(cars.Length);

// Outputs 4
```

Autres façons de créer un tableau

Si vous êtes familier avec C #, vous avez peut-être vu des tableaux créés avec le mot- `new`clé, et peut-être avez-vous également vu des tableaux avec une taille spécifiée. En C#, il existe différentes manières de créer un tableau :

```
// Create an array of four elements, and add values later

string[] cars = new string[4];


// Create an array of four elements and add values right away

string[] cars = new string[4] {"Volvo", "BMW", "Ford", "Mazda"};


// Create an array of four elements without specifying the size

string[] cars = new string[] {"Volvo", "BMW", "Ford", "Mazda"};


// Create an array of four elements, omitting the new keyword, and without
specifying the size

string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

C'est à vous de décider quelle option vous choisissez. Dans notre tutoriel, nous utiliserons souvent la dernière option, car elle est plus rapide et plus facile à lire.

Cependant, notez que si vous déclarez un tableau et que vous l'initialisez ultérieurement, vous devez utiliser le mot- `new`clé :

```
// Declare an array

string[] cars;


// Add values, using new

cars = new string[] {"Volvo", "BMW", "Ford"};


// Add values without using new (this will cause an error)

cars = {"Volvo", "BMW", "Ford"};
```

Boucle dans un tableau

Vous pouvez parcourir les éléments du tableau avec la `for`boucle et utiliser la `Length` propriété pour spécifier combien de fois la boucle doit s'exécuter.

L'exemple suivant affiche tous les éléments du tableau **cars :**

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (int i = 0; i < cars.Length; i++)

{

  Console.WriteLine(cars[i]);

}
```

La boucle foreach

Il existe également une `foreach`boucle, qui est utilisée exclusivement pour parcourir les éléments d'un **tableau** :

```
foreach (type variableName in arrayName)
{
  // code block to be executed
}
```

L'exemple suivant affiche tous les éléments du tableau **cars**`foreach` à l'aide d'une boucle :

```
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
foreach (string i in cars)
{
  Console.WriteLine(i);
}
```

L'exemple ci-dessus peut être lu comme ceci : **pour chaque** `string` élément (appelé **i** - comme dans index) dans **cars** , imprimez la valeur de **i .**

Si vous comparez la `for`boucle et la `foreach`boucle, vous verrez que la `foreach`méthode est plus facile à écrire, qu'elle ne nécessite pas de compteur (en utilisant la `Length`propriété) et qu'elle est plus lisible.

Trier un tableau

Il existe de nombreuses méthodes de tableau disponibles, par exemple `Sort()`, qui trie un tableau par ordre alphabétique ou croissant :

```
// Sort a string
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
Array.Sort(cars);
foreach (string i in cars)
{
  Console.WriteLine(i);
}
```

```
// Sort an int
int[] myNumbers = {5, 1, 8, 9};
Array.Sort(myNumbers);
foreach (int i in myNumbers)
{
  Console.WriteLine(i);
}
```

Espace de noms System.Linq

D'autres méthodes de tableau utiles, telles que `Min`, `Max`et `Sum`, peuvent être trouvées dans l' espace de `System.Linq`noms :

```
using System;
using System.Linq;

namespace MyApplication
{
```

```
  class Program
  {
    static void Main(string[] args)
    {
      int[] myNumbers = {5, 1, 8, 9};
      Console.WriteLine(myNumbers.Max());  // returns the largest value
      Console.WriteLine(myNumbers.Min());  // returns the smallest value
      Console.WriteLine(myNumbers.Sum());  // returns the sum of elements
    }
  }
}
```

Méthodes C#

Une **méthode** est un bloc de code qui ne s'exécute que lorsqu'elle est appelée.

Vous pouvez transmettre des données, appelées paramètres, dans une méthode.

Les méthodes sont utilisées pour effectuer certaines actions, et elles sont également appelées **fonctions** .

Pourquoi utiliser des méthodes ? Pour réutiliser le code : définissez le code une fois et réutilisez-le plusieurs fois.

Créer une méthode

Une méthode est définie avec le nom de la méthode, suivi de parenthèses **()** . C# fournit des méthodes prédéfinies, que vous connaissez déjà, telles que `Main()`, mais vous pouvez également créer vos propres méthodes pour effectuer certaines actions :

Créez une méthode dans la classe Program :

```
class Program
{
  static void MyMethod()
  {
    // code to be executed
  }
}
```

Exemple expliqué

- `MyMethod()`est le nom de la méthode
- `static`signifie que la méthode appartient à la classe Program et non à un objet de la classe Program. Vous en apprendrez plus sur les objets et comment accéder aux méthodes via des objets plus tard dans ce didacticiel.
- `void`signifie que cette méthode n'a pas de valeur de retour. Vous en apprendrez plus sur les valeurs de retour plus loin dans ce chapitre

Remarque : En C#, il est recommandé de commencer par une lettre majuscule lorsque vous nommez des méthodes, car cela facilite la lecture du code.

Appeler une méthode

Pour appeler (exécuter) une méthode, écrivez le nom de la méthode suivi de deux parenthèses **()** et d'un point-virgule **;**

Dans l'exemple suivant, `MyMethod()`sert à imprimer un texte (l'action), lorsqu'elle est appelée :

À l' intérieur `Main()`de , appelez la `myMethod()`méthode :

```
static void MyMethod()
{
  Console.WriteLine("I just got executed!");
}

static void Main(string[] args)
{
  MyMethod();
}

// Outputs "I just got executed!"
```

Une méthode peut être appelée plusieurs fois :

```
static void MyMethod()
{
  Console.WriteLine("I just got executed!");
}
```

```
static void Main(string[] args)
{
  MyMethod();
  MyMethod();
  MyMethod();
}

// I just got executed!
// I just got executed!
// I just got executed!
```

Paramètres et arguments

Les informations peuvent être transmises aux méthodes en tant que paramètre. Les paramètres agissent comme des variables à l'intérieur de la méthode.

Ils sont spécifiés après le nom de la méthode, entre parenthèses. Vous pouvez ajouter autant de paramètres que vous le souhaitez, il suffit de les séparer par une virgule.

L'exemple suivant a une méthode qui prend un **fname**`string` appelé comme paramètre. Lorsque la méthode est appelée, nous transmettons un prénom, qui est utilisé à l'intérieur de la méthode pour imprimer le nom complet :

```
static void MyMethod(string fname)
{
  Console.WriteLine(fname + " Refsnes");
}
```

```
static void Main(string[] args)
{
  MyMethod("Liam");
  MyMethod("Jenny");
  MyMethod("Anja");
}

// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

Lorsqu'un **paramètre** est passé à la méthode, il est appelé **argument** . Ainsi, à partir de l'exemple ci-dessus : `fname`est un **paramètre**`Liam` , tandis que `Jenny`et `Anja`sont des **arguments** .

Paramètres multiples

Vous pouvez avoir autant de paramètres que vous le souhaitez, séparez-les simplement par des virgules :

```
static void MyMethod(string fname, int age)
{
  Console.WriteLine(fname + " is " + age);
}

static void Main(string[] args)
```

```
{
  MyMethod("Liam", 5);
  MyMethod("Jenny", 8);
  MyMethod("Anja", 31);
}

// Liam is 5
// Jenny is 8
// Anja is 31
```

Notez que lorsque vous travaillez avec plusieurs paramètres, l'appel de méthode doit avoir le même nombre d'arguments qu'il y a de paramètres, et les arguments doivent être passés dans le même ordre.

Valeur de paramètre par défaut

Vous pouvez également utiliser une valeur de paramètre par défaut, en utilisant le signe égal (`=`).

Si nous appelons la méthode sans argument, elle utilise la valeur par défaut ("Norway") :

```
static void MyMethod(string country = "Norway")
{
  Console.WriteLine(country);
}

static void Main(string[] args)
```

```
{
  MyMethod("Sweden");
  MyMethod("India");
  MyMethod();
  MyMethod("USA");
}

// Sweden
// India
// Norway
// USA
```

Un paramètre avec une valeur par défaut, est souvent appelé " **paramètre optionnel** ". Dans l'exemple ci-dessus, `country`est un paramètre facultatif et `"Norway"`est la valeur par défaut.

Valeurs de retour

Dans la page précédente , nous avons utilisé le mot- `void`clé dans tous les exemples, ce qui indique que la méthode ne doit pas renvoyer de valeur.

Si vous souhaitez que la méthode renvoie une valeur, vous pouvez utiliser un type de données primitif (tel que `int` ou `double`) au lieu de `void`, et utiliser le `return` mot clé à l'intérieur de la méthode :

```
static int MyMethod(int x)
{
  return 5 + x;
}
```

```
static void Main(string[] args)
{
  Console.WriteLine(MyMethod(3));
}

// Outputs 8 (5 + 3)
```

Cet exemple renvoie la somme des **deux paramètres** d'une méthode :

```
static int MyMethod(int x, int y)
{
  return x + y;
}

static void Main(string[] args)
{
  Console.WriteLine(MyMethod(5, 3));
}

// Outputs 8 (5 + 3)
```

Vous pouvez également stocker le résultat dans une variable (recommandé, car il est plus facile à lire et à maintenir) :

```
static int MyMethod(int x, int y)
{
```

```
  return x + y;
}

static void Main(string[] args)
{
  int z = MyMethod(5, 3);
  Console.WriteLine(z);
}

// Outputs 8 (5 + 3)
```

Arguments nommés

Il est également possible d'envoyer des arguments avec la *key: value*syntaxe.

De cette façon, l'ordre des arguments n'a pas d'importance :

```
static void MyMethod(string child1, string child2, string child3)
{
  Console.WriteLine("The youngest child is: " + child3);
}

static void Main(string[] args)
{
  MyMethod(child3: "John", child1: "Liam", child2: "Liam");
}
```

```
// The youngest child is: John
```

Surcharge de méthode

Avec la **surcharge de méthode** , plusieurs méthodes peuvent avoir le même nom avec des paramètres différents :

```
int MyMethod(int x)

float MyMethod(float x)

double MyMethod(double x, double y)
```

Considérez l'exemple suivant, qui comporte deux méthodes qui additionnent des nombres de types différents :

```
static int PlusMethodInt(int x, int y)

{

  return x + y;

}


static double PlusMethodDouble(double x, double y)

{

  return x + y;

}


static void Main(string[] args)
```

```
{
  int myNum1 = PlusMethodInt(8, 5);
  double myNum2 = PlusMethodDouble(4.3, 6.26);
  Console.WriteLine("Int: " + myNum1);
  Console.WriteLine("Double: " + myNum2);
}
```

Au lieu de définir deux méthodes qui doivent faire la même chose, il vaut mieux en surcharger une.

Dans l'exemple ci-dessous, nous surchargeons la `PlusMethod` méthode pour qu'elle fonctionne à la fois pour `int` et `double`:

```
static int PlusMethod(int x, int y)
{
  return x + y;
}

static double PlusMethod(double x, double y)
{
  return x + y;
}

static void Main(string[] args)
{
  int myNum1 = PlusMethod(8, 5);
  double myNum2 = PlusMethod(4.3, 6.26);
```

```
  Console.WriteLine("Int: " + myNum1);

  Console.WriteLine("Double: " + myNum2);

}
```

Remarque : plusieurs méthodes peuvent avoir le même nom tant que le nombre et/ou le type de paramètres sont différents.

C# Classes

C# - Qu'est-ce que la POO ?

OOP signifie Programmation Orientée Objet.

La programmation procédurale consiste à écrire des procédures ou des méthodes qui effectuent des opérations sur les données, tandis que la programmation orientée objet consiste à créer des objets contenant à la fois des données et des méthodes.

La programmation orientée objet présente plusieurs avantages par rapport à la programmation procédurale :

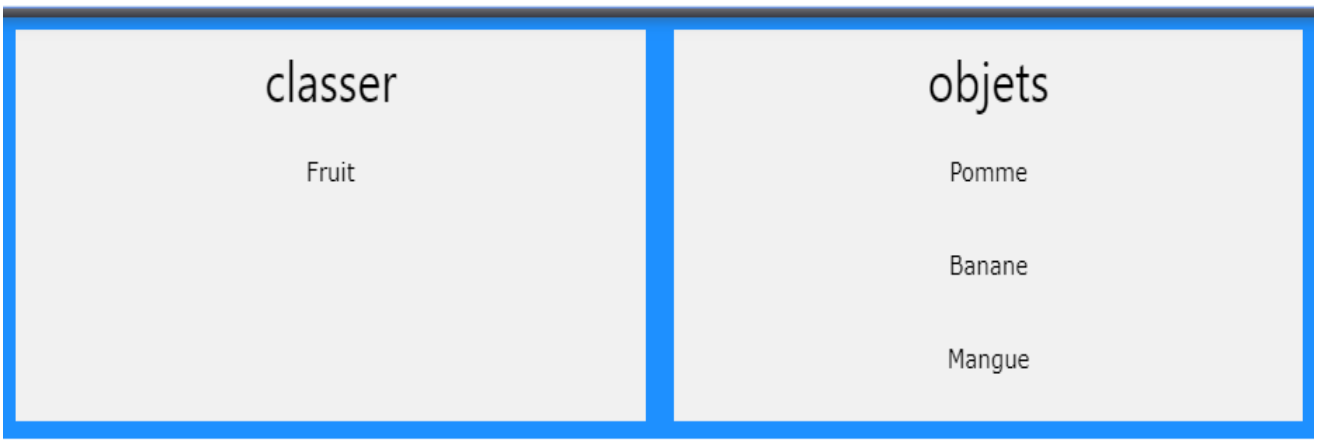
- La POO est plus rapide et plus facile à exécuter
- La POO fournit une structure claire pour les programmes
- La POO aide à garder le code C# DRY "Ne vous répétez pas", et facilite la maintenance, la modification et le débogage du code
- La POO permet de créer des applications entièrement réutilisables avec moins de code et un temps de développement plus court

Conseil : Le principe "Ne vous répétez pas" (DRY) consiste à réduire la répétition du code. Vous devez extraire les codes communs à l'application, les placer à un seul endroit et les réutiliser au lieu de les répéter.

C# - Que sont les classes et les objets ?

Les classes et les objets sont les deux principaux aspects de la programmation orientée objet.

Regardez l'illustration suivante pour voir la différence entre la classe et les objets :



Un autre exemple:

Ainsi, une classe est un modèle pour les objets et un objet est une instance d'une classe.

Lorsque les objets individuels sont créés, ils héritent de toutes les variables et méthodes de la classe.

Vous en apprendrez beaucoup plus sur les classes et les objets dans le chapitre suivant.

Classes et objets

Vous avez appris du chapitre précédent que C# est un langage de programmation orienté objet.

Tout en C # est associé à des classes et des objets, ainsi qu'à ses attributs et méthodes. Par exemple : dans la vraie vie, une voiture est un objet. La voiture a des **attributs** , tels que le poids et la couleur, et **des méthodes** , telles que la conduite et le freinage.

Une classe est comme un constructeur d'objets ou un "plan" pour créer des objets.

Créer une classe

Pour créer une classe, utilisez le `class`mot clé :

Créez une classe nommée " `Car`" avec une variable`color` :

```
class Car

{

  string color = "red";

}
```

Lorsqu'une variable est déclarée directement dans une classe, elle est souvent appelée **champ** (ou attribut).

Ce n'est pas obligatoire, mais c'est une bonne pratique de commencer par une première lettre majuscule lorsque vous nommez des classes. De plus, il est courant que le nom du fichier C # et la classe correspondent, car cela rend notre code organisé. Cependant, ce n'est pas obligatoire (comme en Java).

Créer un objet

Un objet est créé à partir d'une classe. Nous avons déjà créé la classe nommée `Car`, nous pouvons donc maintenant l'utiliser pour créer des objets.

Pour créer un objet de `Car`, spécifiez le nom de la classe, suivi du nom de l'objet, et utilisez le mot-clé `new`:

Créez un objet appelé " `myObj`" et utilisez-le pour imprimer la valeur de `color` :

```
class Car
{
  string color = "red";

  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
  }
}
```

Notez que nous utilisons la syntaxe à points (`.`) pour accéder aux variables/champs à l'intérieur d'une classe (`myObj.color`). Vous en apprendrez plus sur les champs dans le chapitre suivant.

Objets multiples

Vous pouvez créer plusieurs objets d'une même classe :

Créez deux objets de `Car`

```
class Car
{
  string color = "red";
  static void Main(string[] args)
  {
    Car myObj1 = new Car();
    Car myObj2 = new Car();
    Console.WriteLine(myObj1.color);
    Console.WriteLine(myObj2.color);
  }
}
```

Utilisation de plusieurs classes

Vous pouvez également créer un objet d'une classe et y accéder dans une autre classe. Ceci est souvent utilisé pour une meilleure organisation des classes (une classe contient tous les champs et méthodes, tandis que l'autre classe contient la `Main()`méthode (code à exécuter)).

- prog2.cs
- prog.cs

prog2.cs

```
class Car

{

  public string color = "red";

}
```

prog.cs

```
class Program

{

  static void Main(string[] args)

  {

    Car myObj = new Car();

    Console.WriteLine(myObj.color);

  }

}
```

Avez-vous remarqué le `public`mot-clé ? C'est ce qu'on appelle un **modificateur d'accès** , qui spécifie que la `color`variable/le champ de `Car`est également accessible pour d'autres classes, telles que `Program`.

Vous en apprendrez beaucoup plus sur **les modificateurs d'accès** et les **classes/objets** dans les prochains chapitres.

Membres du groupe

Les champs et les méthodes à l'intérieur des classes sont souvent appelés "membres de classe":

Créez une `Car`classe avec trois membres de classe : **deux champs** et **une méthode** .

```
// The class
class MyClass
{
  // Class members
  string color = "red";        // field
  int maxSpeed = 200;          // field
  public void fullThrottle()   // method
  {
    Console.WriteLine("The car is going as fast as it can!");
  }
}
```

Des champs

Dans le chapitre précédent, vous avez appris que les variables à l'intérieur d'une classe sont appelées des champs, et que vous pouvez y accéder en créant un objet de la classe, et en utilisant la syntaxe à point (`.`).

L'exemple suivant créera un objet de la `Car`classe, avec le nom `myObj`. Puis on imprime la valeur des champs `color` et `maxSpeed`:

```
class Car
{
  string color = "red";
  int maxSpeed = 200;

  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
    Console.WriteLine(myObj.maxSpeed);
  }
}
```

Vous pouvez également laisser les champs vides, et les modifier lors de la création de l'objet :

```
class Car
{
  string color;
  int maxSpeed;

  static void Main(string[] args)
  {
    Car myObj = new Car();
    myObj.color = "red";
    myObj.maxSpeed = 200;
```

```
    Console.WriteLine(myObj.color);
    Console.WriteLine(myObj.maxSpeed);
  }
}
```

Ceci est particulièrement utile lors de la création de plusieurs objets d'une même classe :

```
class Car
{
  string model;
  string color;
  int year;

  static void Main(string[] args)
  {
    Car Ford = new Car();
    Ford.model = "Mustang";
    Ford.color = "red";
    Ford.year = 1969;

    Car Opel = new Car();
    Opel.model = "Astra";
    Opel.color = "white";
    Opel.year = 2005;
```

```
    Console.WriteLine(Ford.model);
    Console.WriteLine(Opel.model);
  }
}
```

Constructeurs

Un constructeur est une **méthode spéciale** utilisée pour initialiser des objets. L'avantage d'un constructeur, c'est qu'il est appelé lorsqu'un objet d'une classe est créé. Il peut être utilisé pour définir des valeurs initiales pour les champs :

Créez un constructeur :

```
// Create a Car class
class Car
{
  public string model;  // Create a field

  // Create a class constructor for the Car class
  public Car()
  {
    model = "Mustang"; // Set the initial value for model
  }

  static void Main(string[] args)
  {
    Car Ford = new Car();  // Create an object of the Car Class (this will
call the constructor)
```

```
    Console.WriteLine(Ford.model);  // Print the value of model
  }
}


// Outputs "Mustang"
```

Notez que le nom du constructeur doit **correspondre au nom de la classe** et qu'il ne peut pas avoir de **type de retour** (comme `void`ou `int`).

Notez également que le constructeur est appelé lorsque l'objet est créé.

Toutes les classes ont des constructeurs par défaut : si vous ne créez pas de constructeur de classe vous-même, C# en crée un pour vous. Cependant, vous ne pouvez pas définir de valeurs initiales pour les champs.

Les constructeurs gagnent du temps ! Jetez un œil au dernier exemple sur cette page pour vraiment comprendre pourquoi.

Paramètres du constructeur

Les constructeurs peuvent également prendre des paramètres, qui sont utilisés pour initialiser les champs.

L'exemple suivant ajoute un `string modelName`paramètre au constructeur. À l'intérieur du constructeur, nous définissons `model`sur `modelName`(`model=modelName`). Lorsque nous appelons le constructeur, nous passons un paramètre au constructeur (`"Mustang"`), qui définira la valeur de `model`à `"Mustang"`:

```
class Car
{
  public string model;
```

```
  // Create a class constructor with a parameter
  public Car(string modelName)
  {
    model = modelName;
  }

  static void Main(string[] args)
  {
    Car Ford = new Car("Mustang");
    Console.WriteLine(Ford.model);
  }
}

// Outputs "Mustang"
```

Vous pouvez avoir autant de paramètres que vous le souhaitez :

```
class Car
{
  public string model;
  public string color;
  public int year;

  // Create a class constructor with multiple parameters
  public Car(string modelName, string modelColor, int modelYear)
  {
```

```
    model = modelName;
    color = modelColor;
    year = modelYear;
  }

  static void Main(string[] args)
  {
    Car Ford = new Car("Mustang", "Red", 1969);
    Console.WriteLine(Ford.color + " " + Ford.year + " " + Ford.model);
  }
}



// Outputs Red 1969 Mustang
```

Conseil : Tout comme les autres méthodes, les constructeurs peuvent être **surchargés** en utilisant un nombre différent de paramètres.

Les constructeurs gagnent du temps

Si vous considérez l'exemple du chapitre précédent, vous remarquerez que les constructeurs sont très utiles, car ils aident à réduire la quantité de code :

Sans constructeur :

```
{
  static void Main(string[] args)
  {
    Car Ford = new Car();
    Ford.model = "Mustang";
    Ford.color = "red";
    Ford.year = 1969;

    Car Opel = new Car();
    Opel.model = "Astra";
    Opel.color = "white";
    Opel.year = 2005;

    Console.WriteLine(Ford.model);
    Console.WriteLine(Opel.model);
  }
}
```

Avec constructeur :

prog.cs

```
class Program
{
  static void Main(string[] args)
  {
    Car Ford = new Car("Mustang", "Red", 1969)
    Car Opel = new Car("Astra", "White", 2005)

    Console.WriteLine(Ford.model);
    Console.WriteLine(Opel.model);
  }
}
```

Modificateurs d'accès

À présent, vous connaissez bien le `public`mot-clé qui apparaît dans bon nombre de nos exemples :

```
public string color;
```

Le `public`mot-clé est un **modificateur d'accès** , qui est utilisé pour définir le niveau d'accès/la visibilité des classes, des champs, des méthodes et des propriétés.

C# a les modificateurs d'accès suivants :

Modifier	Description
`public`	The code is accessible for all classes
`private`	The code is only accessible within the same class
`protected`	The code is accessible within the same class, or in a class that is inherited from that class. You will learn more about inheritance in a later chapter
`internal`	The code is only accessible within its own assembly, but not from another assembly. You will learn more about this in a later chapter

Il existe également deux combinaisons : `protected internal`et `private protected`.

Pour l'instant, concentrons-nous sur `public`et sur `private`les modificateurs.

Modificateur privé

Si vous déclarez un champ avec un `private`modificateur d'accès, il n'est accessible qu'au sein de la même classe :

```
class Car


{ { } }
  private string model = "Mustang";


  static void Main(string[] args)


    Car myObj = new Car();

    Console.WriteLine(myObj.model);
```

La sortie sera :

```
Mustang
```

Si vous essayez d'y accéder en dehors de la classe, une erreur se produira :

```
class Car
{
  private string model = "Mustang";
}

class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.model);
  }
}
```

La sortie sera :

```
'Car.model' is inaccessible due to its protection level
The field 'Car.model' is assigned but its value is never used
```

Propriétés C# (Get et Set)

Propriétés et encapsulation

Avant de commencer à expliquer les propriétés, vous devez avoir une compréhension de base de " **Encapsulation** ".

La signification d' **Encapsulation** est de s'assurer que les données "sensibles" sont cachées aux utilisateurs. Pour y parvenir, vous devez :

- déclarer des champs/variables comme`private`
- fournir `public get` et `set`méthodes, via **properties** , pour accéder et mettre à jour la valeur d'un `private` champ
-

Propriétés

- Vous avez appris du chapitre précédent que `private`les variables ne sont accessibles qu'au sein d'une même classe (une classe extérieure n'y a pas accès). Cependant, nous devons parfois y accéder - et cela peut être fait avec des propriétés.
- Une propriété est comme une combinaison d'une variable et d'une méthode, et elle a deux méthodes : une `get`et une `set`méthode :

```
class Person

{

  private string name; // field


  public string Name   // property

  {

    get { return name; }   // get method

    set { name = value; }  // set method

  }

}
```

Exemple expliqué

La `Name`propriété est associée au `name`champ. Il est recommandé d'utiliser le même nom pour la propriété et le champ privé, mais avec une première lettre en majuscule.

La `get`méthode renvoie la valeur de la variable `name`.

La `set`méthode affecte un `value`à la `name`variable. Le `value`mot-clé représente la valeur que nous attribuons à la propriété.

Si vous ne le comprenez pas entièrement, jetez un œil à l'exemple ci-dessous.

Nous pouvons maintenant utiliser la `Name`propriété pour accéder et mettre à jour le `private`champ de la `Person`classe :

```
class Person
{
  private string name; // field
  public string Name   // property
  {
    get { return name; }
    set { name = value; }
  }
}

class Program
{
  static void Main(string[] args)
  {
```

```
    Person myObj = new Person();
    myObj.Name = "Liam";
    Console.WriteLine(myObj.Name);
  }
}
```

La sortie sera :

```
Liam
```

Héritage C#

Héritage (classe dérivée et de base)

En C#, il est possible d'hériter des champs et des méthodes d'une classe à une autre. Nous regroupons le "concept d'héritage" en deux catégories :

- **Classe dérivée** (enfant) - la classe qui hérite d'une autre classe
- **Classe de base** (parent) - la classe héritée de

Pour hériter d'une classe, utilisez le `:`symbole.

Dans l'exemple ci-dessous, la `Car`classe (enfant) hérite des champs et des méthodes de la `Vehicle`classe (parent) :

```
class Vehicle  // base class (parent)
{
  public string brand = "Ford";  // Vehicle field
  public void honk()             // Vehicle method
  {
    Console.WriteLine("Tuut, tuut!");
```

```
  }
}


class Car : Vehicle  // derived class (child)
{
  public string modelName = "Mustang";  // Car field
}


class Program
{
  static void Main(string[] args)
  {
    // Create a myCar object
    Car myCar = new Car();


    // Call the honk() method (From the Vehicle class) on the myCar object
    myCar.honk();


    // Display the value of the brand field (from the Vehicle class) and
the value of the modelName from the Car class
    Console.WriteLine(myCar.brand + " " + myCar.modelName);
  }
}
```

Pourquoi et quand utiliser "Héritage" ?

- C'est utile pour la réutilisabilité du code : réutilisez les champs et les méthodes d'une classe existante lorsque vous créez une nouvelle classe.

Astuce : Jetez également un œil au chapitre suivant, Polymorphisme , qui utilise des méthodes héritées pour effectuer différentes tâches.

Le mot-clé scellé

Si vous ne voulez pas que d'autres classes héritent d'une classe, utilisez le mot- `sealed`clé :

Si vous essayez d'accéder à une `sealed`classe, C# générera une erreur :

```
sealed class Vehicle
{
  ...
}

class Car : Vehicle
{
  ...
}
```

Le message d'erreur ressemblera à ceci :

```
'Car': cannot derive from sealed type 'Vehicle'
```

Polymorphisme C#

Polymorphisme et méthodes de substitution

Le polymorphisme signifie "plusieurs formes", et il se produit lorsque nous avons de nombreuses classes liées les unes aux autres par héritage.

Comme nous l'avons précisé dans le chapitre précédent; **L'héritage** nous permet d'hériter des champs et des méthodes d'une autre classe. **Le polymorphisme** utilise ces méthodes pour effectuer différentes tâches. Cela nous permet d'effectuer une même action de différentes manières.

Par exemple, pensez à une classe de base appelée `Animal`qui a une méthode appelée `animalSound()`. Les classes dérivées d'animaux pourraient être des cochons, des chats, des chiens, des oiseaux - et ils ont aussi leur propre implémentation d'un son d'animal (le cochon ronfle, et le chat miaule, etc.) :

```
class Animal  // Base class (parent)
{
  public void animalSound()
  {
    Console.WriteLine("The animal makes a sound");
  }
}


class Pig : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The pig says: wee wee");
```

```
  }
}


class Dog : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The dog says: bow wow");
  }
}
```

Rappelez-vous du chapitre Héritage que nous utilisons le `:`symbole pour hériter d'une classe.

Nous pouvons maintenant créer des objets et appeler la `Pig`méthode sur les deux : `DoganimalSound()`

```
class Animal  // Base class (parent)
{
  public void animalSound()
  {
    Console.WriteLine("The animal makes a sound");
  }
}


class Pig : Animal  // Derived class (child)
{
```

```
  public void animalSound()
  {
    Console.WriteLine("The pig says: wee wee");
  }
}

class Dog : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The dog says: bow wow");
  }
}

class Program
{
  static void Main(string[] args)
  {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object

    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
```

```
  }
}
```

La sortie sera :

```
The animal makes a sound
The animal makes a sound
The animal makes a sound
```

Pas la sortie que je recherchais

La sortie de l'exemple ci-dessus n'était probablement pas celle que vous attendiez. En effet, la méthode de classe de base remplace la méthode de classe dérivée, lorsqu'elles partagent le même nom.

Cependant, C# fournit une option pour remplacer la méthode de classe de base, en ajoutant le mot- `virtual`clé à la méthode à l'intérieur de la classe de base, et en utilisant le mot- `override`clé pour chaque méthode de classe dérivée :

```
class Animal  // Base class (parent)
{
  public virtual void animalSound()
  {
    Console.WriteLine("The animal makes a sound");
  }
}


class Pig : Animal  // Derived class (child)
{
  public override void animalSound()
  {
```

```
    Console.WriteLine("The pig says: wee wee");
  }
}

class Dog : Animal  // Derived class (child)
{
  public override void animalSound()
  {
    Console.WriteLine("The dog says: bow wow");
  }
}

class Program
{
  static void Main(string[] args)
  {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object

    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

La sortie sera :

```
The animal makes a sound
The pig says: wee wee
The dog says: bow wow
```

Pourquoi et quand utiliser "Héritage" et "Polymorphisme" ?

- C'est utile pour la réutilisabilité du code : réutilisez les champs et les méthodes d'une classe existante lorsque vous créez une nouvelle classe.

C# Abstraction

Classes abstraites et méthodes

L' abstraction de données est le processus consistant à masquer certains détails et à ne montrer que les informations essentielles à l'utilisateur. L'abstraction peut être réalisée avec **des classes abstraites** ou **des interfaces** (sur lesquelles vous en apprendrez plus dans le chapitre suivant).

Le `abstract`mot-clé est utilisé pour les classes et les méthodes :

- **Classe abstraite :** est une classe restreinte qui ne peut pas être utilisée pour créer des objets (pour y accéder, elle doit être héritée d'une autre classe).

- **Méthode abstraite :** ne peut être utilisée que dans une classe abstraite et n'a pas de corps. Le corps est fourni par la classe dérivée (héritée de).

Une classe abstraite peut avoir à la fois des méthodes abstraites et régulières :

```
abstract class Animal

{

  public abstract void animalSound();

  public void sleep()
```

```
  {
    Console.WriteLine("Zzz");
  }
}
```

A partir de l'exemple ci-dessus, il n'est pas possible de créer un objet de la classe Animal :

```
Animal myObj = new Animal(); // Will generate an error (Cannot create an instance of the abstract class or interface 'Animal')
```

Pour accéder à la classe abstraite, elle doit être héritée d'une autre classe. Convertissons la classe Animal que nous avons utilisée dans le chapitre Polymorphisme en une classe abstraite.

Rappelez-vous du chapitre Héritage que nous utilisons le `:`symbole pour hériter d'une classe et que nous utilisons le mot- `override`clé pour remplacer la méthode de classe de base.

```
// Abstract class
abstract class Animal
{
  // Abstract method (does not have a body)
  public abstract void animalSound();
  // Regular method
  public void sleep()
  {
    Console.WriteLine("Zzz");
```

```
  }
}


// Derived class (inherit from Animal)
class Pig : Animal
{
  public override void animalSound()
  {
    // The body of animalSound() is provided here
    Console.WriteLine("The pig says: wee wee");
  }
}


class Program
{
  static void Main(string[] args)
  {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();  // Call the abstract method
    myPig.sleep();  // Call the regular method
  }
}
```

Pourquoi et quand utiliser des classes et des méthodes abstraites ?

Pour assurer la sécurité, masquez certains détails et affichez uniquement les détails importants d'un objet.

Remarque : L'abstraction peut également être réalisée avec Interfaces , dont vous en apprendrez plus dans le chapitre suivant.

Interface C#

Interfaces

Une autre façon de réaliser l'abstraction en C # consiste à utiliser des interfaces.

An `interface`est une " **classe complètement abstraite** ", qui ne peut contenir que des méthodes et des propriétés abstraites (avec des corps vides) :

```
// interface

interface Animal

{

  void animalSound(); // interface method (does not have a body)

  void run(); // interface method (does not have a body)

}
```

Il est considéré comme une bonne pratique de commencer par la lettre "I" au début d'une interface, car cela permet à vous-même et aux autres de vous souvenir plus facilement qu'il s'agit d'une interface et non d'une classe.

Par défaut, les membres d'une interface sont `abstract`et `public`.

Remarque : Les interfaces peuvent contenir des propriétés et des méthodes, mais pas des champs.

Pour accéder aux méthodes d'interface, l'interface doit être "implémentée" (un peu comme héritée) par une autre classe. Pour implémenter une interface, utilisez le `:` symbole (comme pour l'héritage). Le corps de la méthode d'interface est fourni par la classe "implement". Notez que vous n'êtes pas obligé d'utiliser le mot- `override` clé lors de l'implémentation d'une interface :

```
// Interface
interface IAnimal
{
  void animalSound(); // interface method (does not have a body)
}

// Pig "implements" the IAnimal interface
class Pig : IAnimal
{
  public void animalSound()
  {
    // The body of animalSound() is provided here
    Console.WriteLine("The pig says: wee wee");
  }
}

class Program
{
  static void Main(string[] args)
  {
    Pig myPig = new Pig();  // Create a Pig object
    myPig.animalSound();
  }
}
```

Remarques sur les interfaces :

- Comme **les classes abstraites** , les interfaces **ne peuvent pas** être utilisées pour créer des objets (dans l'exemple ci-dessus, il n'est pas possible de créer un objet "IAnimal" dans la classe Program)
- Les méthodes d'interface n'ont pas de corps - le corps est fourni par la classe "implémenter"
- Lors de l'implémentation d'une interface, vous devez surcharger toutes ses méthodes
- Les interfaces peuvent contenir des propriétés et des méthodes, mais pas des champs/variables
- Les membres de l'interface sont par défaut `abstract`et `public`
- Une interface ne peut pas contenir de constructeur (car elle ne peut pas être utilisée pour créer des objets)

Pourquoi et quand utiliser les interfaces ?

1) Pour assurer la sécurité - masquez certains détails et affichez uniquement les détails importants d'un objet (interface).

2) C# ne prend pas en charge "l'héritage multiple" (une classe ne peut hériter que d'une classe de base). Cependant, cela peut être réalisé avec des interfaces, car la classe peut **implémenter** plusieurs interfaces. **Remarque :** Pour implémenter plusieurs interfaces, séparez-les par une virgule (voir l'exemple ci-dessous).

Interfaces multiples

Pour implémenter plusieurs interfaces, séparez-les par une virgule :

```
interface IFirstInterface

{

  void myMethod(); // interface method

}
```

```
interface ISecondInterface
{
  void myOtherMethod(); // interface method
}

// Implement multiple interfaces
class DemoClass : IFirstInterface, ISecondInterface
{
  public void myMethod()
  {
    Console.WriteLine("Some text..");
  }
  public void myOtherMethod()
  {
    Console.WriteLine("Some other text...");
  }
}

class Program
{
  static void Main(string[] args)
  {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
```

```
  }
}
```

Énumérations C#

An `enum`est une "classe" spéciale qui représente un groupe de **constantes** (variables non modifiables/en lecture seule).

Pour créer un `enum`, utilisez le `enum`mot clé (au lieu de class ou interface) et séparez les éléments enum par une virgule :

```
enum Level
{
  Low,
  Medium,
  High
}
```

Vous pouvez accéder aux `enum`éléments avec la syntaxe à **points :**

```
Level myVar = Level.Medium;
Console.WriteLine(myVar);
```

Enum est l'abréviation de "énumérations", ce qui signifie "spécifiquement répertorié".

Énumération à l'intérieur d'une classe

Vous pouvez également avoir un `enum`intérieur d'une classe :

```
class Program
{
  enum Level
  {
    Low,
    Medium,
    High
  }
  static void Main(string[] args)
  {
    Level myVar = Level.Medium;
    Console.WriteLine(myVar);
  }
}
```

La sortie sera :

```
Medium
```

Fichiers C#

Travailler avec des fichiers

La `File`classe de l' espace de `System.IO`noms, nous permet de travailler avec des fichiers :

```
using System.IO;  // include the System.IO namespace


File.SomeFileMethod();  // use the file class with methods
```

La `File`classe dispose de nombreuses méthodes utiles pour créer et obtenir des informations sur les fichiers. Par exemple:

Method	Description
`AppendText()`	Appends text at the end of an existing file
`Copy()`	Copies a file
`Create()`	Creates or overwrites a file
`Delete()`	Deletes a file
`Exists()`	Tests whether the file exists
`ReadAllText()`	Reads the contents of a file
`Replace()`	Replaces the contents of a file with the contents of another file
`WriteAllText()`	Creates a new file and writes the contents to it. If the file already exists, it will be overwritten.

Pour une liste complète des méthodes File, accédez à Microsoft .Net File Class Reference .

Écrire dans un fichier et le lire

Dans l'exemple suivant, nous utilisons la `WriteAllText()`méthode pour créer un fichier nommé "filename.txt" et y écrire du contenu. Ensuite, nous utilisons la `ReadAllText()` méthode pour lire le contenu du fichier :

```
using System.IO;  // include the System.IO namespace


string writeText = "Hello World!";  // Create a text string

File.WriteAllText("filename.txt", writeText);  // Create a file and write
the content of writeText to it


string readText = File.ReadAllText("filename.txt");  // Read the contents
of the file

Console.WriteLine(readText);  // Output the content
```

La sortie sera :

```
Hello World!
```

Exceptions C# - Try..Catch

Exception C#

Lors de l'exécution du code C#, différentes erreurs peuvent se produire : erreurs de codage faites par le programmeur, erreurs dues à une mauvaise saisie ou à d'autres choses imprévisibles.

Lorsqu'une erreur se produit, C# s'arrête normalement et génère un message d'erreur. Le terme technique pour cela est : C# lèvera une **exception** (lancera une erreur).

C# essayer et attraper

L' `try`instruction vous permet de définir un bloc de code à tester pour les erreurs pendant son exécution.

L' `catch`instruction vous permet de définir un bloc de code à exécuter, si une erreur se produit dans le bloc try.

Les mots-clés `try`et `catch`vont par paires :

```
try
{
  //  Block of code to try
}
catch (Exception e)
{
  //  Block of code to handle errors
}
```

Prenons l'exemple suivant, où nous créons un tableau de trois entiers :

Cela générera une erreur, car **myNumbers[10]** n'existe pas.

```
int[] myNumbers = {1, 2, 3};
Console.WriteLine(myNumbers[10]); // error!
```

Le message d'erreur ressemblera à ceci :

```
System.IndexOutOfRangeException: 'Index was outside the bounds of the array.'
```

Si une erreur se produit, nous pouvons utiliser `try...catch` pour intercepter l'erreur et exécuter du code pour la gérer.

Dans l'exemple suivant, nous utilisons la variable à l'intérieur du bloc catch (`e`) avec la `Message`propriété intégrée, qui génère un message décrivant l'exception :

```
try
{
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]);
}
catch (Exception e)
{
  Console.WriteLine(e.Message);
}
```

La sortie sera :

```
Index was outside the bounds of the array.
```

Vous pouvez également générer votre propre message d'erreur :

```
try
{
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]);
}
catch (Exception e)
{
  Console.WriteLine("Something went wrong.");
}
```

La sortie sera :

```
Something went wrong.
```

Pour terminer

L' `finally`instruction vous permet d'exécuter du code, après `try...catch`, quel que soit le résultat :

```
try
{
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]);
}
catch (Exception e)
{
  Console.WriteLine("Something went wrong.");
}
finally
{
  Console.WriteLine("The 'try catch' is finished.");
}
```

La sortie sera :

```
Something went wrong.
The 'try catch' is finished.
```

Le mot-clé lancer

L' `throw`instruction vous permet de créer une erreur personnalisée.

L' `throw`instruction est utilisée avec une **classe d'exception** . Il existe de nombreuses classes d'exceptions disponibles en

C# : `ArithmeticException`, `FileNotFoundException`, `IndexOutOfRangeException`, `TimeOutException`, etc :

```
static void checkAge(int age)
{
  if (age < 18)
  {
    throw new ArithmeticException("Access denied - You must be at least 18 years old.");
  }
  else
  {
    Console.WriteLine("Access granted - You are old enough!");
  }
}

static void Main(string[] args)
{
  checkAge(15);
}
```

Le message d'erreur affiché dans le programme sera :

```
System.ArithmeticException: 'Access denied - You must be at least 18 years old.'
```

Si `age`était 20, vous n'obtiendriez **pas** d'exception :

```
checkAge(20);
```

La sortie sera :

Access granted - You are old enough!

Games Made With C